User-defined Functions

Need for User-defined Functions

- It is possible to code any program utilizing only **main()**function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing and maintaining becomes difficult.
- If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit.
- The independently coded programs are called ***subprograms*** that are much easier to understand, debug and test. In C, such subprograms are referred to as 'subprograms'.
- In many situations, certain types of operations or calculations are repeated at many points throughout a program. At such times, we can design a function that can be called and used whenever required. This saves both time and space.
- This approach is also known as "Modular Programming" which results in a no. of advantages:

Modular Programming

- Modular programming is the process of subdividing a computer program into separate sub-programs called ***modules*** that are separately named and individually callable *program units*.
- These modules are carefully integrated to become a software system that satisfies the system requirements.
- It is a "divide-and-conquer" approach of problem solving.
- Modules are identified and designed such that they can be organized into a top-down hierarchical structure.
- In C, each module refers to a function that is responsible for a single task.

Characteristics of Modular Programming

- Each module can do only one thing.
- Communication between modules is allowed only by a calling module.
- A module can be called by one and only one higher module.
- No communication can take place directly between modules.
- All modules are designed as *single-entry, single exit* systems using control structures.

Advantages of Modular Programming

1. It facilitates top-down modular programming. In this programming style, the high level logic of the overall program is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places.
3. It is easy to locate and isolate a faulty function.
4. A function may be used by many other programs.

Elements of User-defined functions

- A user-defined function has three elements:
  - Function Definition
  - Function Call
  - Function Declaration

1. <u>Function Definition:</u>

- **The function definition** is an independent program module that is specially written to implement to the requirements of the function.
- A function definition, also known as function implementation shall include the following elements
  1. Function type
  2. Function name
  3. List of parameters
  4. Local variable declarations
  5. Function statements
  6. A return statement
- All the six elements are grouped into two parts; namely,
  - **Function header (First three elements)**
  - **Function body (Second three elements)**

- **General Format of a Function Definition**

```
function_type function_name(parameter_list)
{
        local variable declarations;
        executable statement1;
        executable statement2;
        …
        return statement;

}
```

- The first line **function_type function_name(parameter_list)** is known as ***function header*** and the statements within the opening and closing braces constitute the ***function body.***

Function Header

- The function header consists of three parts: function type, function name and list of parameter.

1. Function Type (Return type):

- The function type specifies the type of value (like float or double) that the function is expected to return to the calling program.
- If the return type is not explicitly specified, C will assume that it is an integer type.
- If the function is not returning anything, we need to specify the return type as **void.**

2. Function Name:

- The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function.

3. List of Parameters:

- The parameter list declares the variables that will receive the data sent by the calling program.
- They serve as input data to the function to carry out the specified task.
- Since they represent actual input values, they are often called as "**formal**" **parameters** or **arguments**.

Example : void main(){ .... }
float mul (float x, float y) { .... }
int sum (int a, int b) { .... }

- A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword **void** between the parentheses

Actual Vs Formal Parameters

Actual Parameters	Formal Parameters
The arguments listed in the **function calling** statement are referred to as **actual arguments**.	The arguments used in the **function declaration** are referred as **formal arguments**.
They are the actual values passed to a function to compute a value or to perform a task.	They are simply formal variables that accept or receive the values supplied by the calling program.

Function Body

- The function body contains the declarations and statements necessary for performing the required task.
- The function body is enclosed in braces, contains three parts, in the order given below:
  - **Local variable declaration**: Local variable declarations are statements that specify the variables needed by the function.
  - **Function Statements:** Function statements are statements that perform the task of the function.
  - **Return Statement:** A return statement is a statement that returns the value evaluated by the function to the calling program. If a function does not return any value, one can omit the **return** statement. While it is possible to pass the called function any number of values, the called function can only return one value per call, at the most.

    Syntax: return; or return(expression);

```
float mul(float x, float y)
{
      float result;           //local variable declaration
      result = x * y;         //function statement
      return (result);        //returns the result.
}
```

Note:

- When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a **void** return.
- A *local variable* is a variable that is defined inside a function and used without having any role in the communication between the functions.

2. Function Call:

- In order to use the function we need to invoke it at a required place in the program. This is known as the **function call**.
- A function can be called using the function name followed by a list of actual parameters(arguments), if any, enclosed in { }.

```
int sum(int, int); //declaration       //function definition
void main()                            int sum(int x, int y)//formal args
{                                      {
      int a=5, b=6,ans;                      int val;
                                             val = x +y;
```

```
        //calling function (actual args)
        ans = sum(a , b);                              return val;
        printf("Answer : %d",ans);          }
}
```

- When the compiler encounters a function call, the control is transferred to the function sum();. This function is executed line by line and a value is returned as the **return** statement is encountered. This value is assigned to **ans.**
- In a function call, the function name is the operand and the parentheses(…) which contains the actual parameters is the operator.
- The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note:
1. If the actual parameters are more than formal parameters, the extra actual arguments will be discarded.
2. If the actual are less than the formal arguments, the unmatched formal arguments will be initialized to some garbage values.
3. Any mismatch in datatypes may also result in some garbage values.

3. Function Declaration:

- The program or a function that has called a function is referred to as the **calling function or calling program**. The calling program should declare any function that is to be used later in the program. This is known as the **function declaration.**
- A function declaration consists of four parts. They are,
  - Function type
  - Function name
  - Parameter list
  - Terminating semicolon
- The declaration of a function is known as **function prototype**. The function prototype is coded in the following format, **function_type** function_name(parameter list);
  Example: int mul(int m, int n); // Function prototype

- The prototype declaration may be placed in two places in a program:
  - **Above all the functions (including main)** – When we place the declaration above all the functions , the prototype is referred to as a *global prototype.*
  - Inside a function definition – When we place the declaration inside the local declaration section, the prototype is called a *local prototype*.

Why Prototype is required?:
- If a function has not been declared before it is used, C will assume that is details available at the time of linking.
- Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program.
- Hence, we must always include the prototype declarations.

Nesting of Functions
- In C, each function can contain one or more than one function in it. There is no limit as to how deeply functions can be nested. Consider the following example,

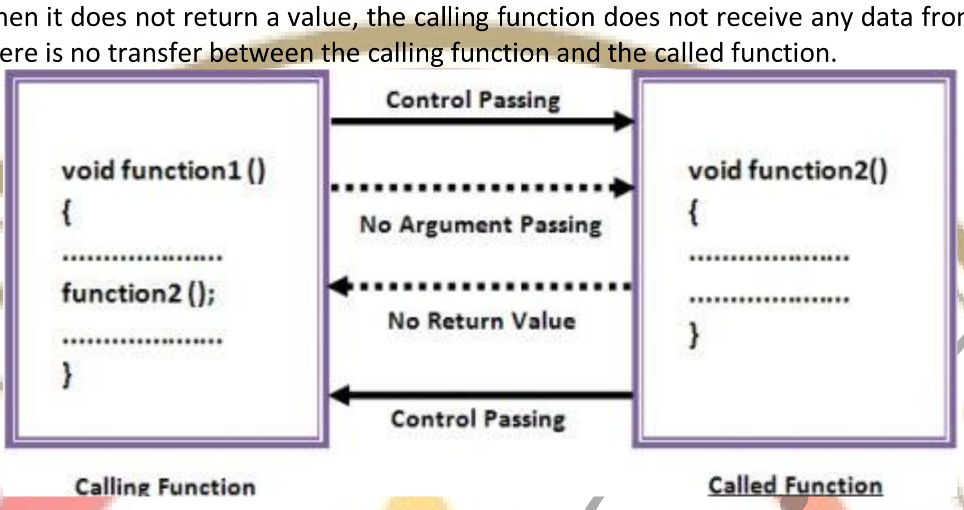
```
void main()
{
        function01();
}
function01()
{
        function02();
}
function02()
{
        ……………;
}
```

Category of Functions

- A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:
    1. Functions with no arguments and no return values
    2. Functions with arguments and no return values
    3. Functions with arguments and one return values
    4. Functions with no arguments but a return value
    5. Functions that return multiple values

Functions with no arguments and no return values

- When a function has no arguments, it does not receive any data from the calling function.
- Similarly, when it does not return a value, the calling function does not receive any data from the called function. There is no transfer between the calling function and the called function.



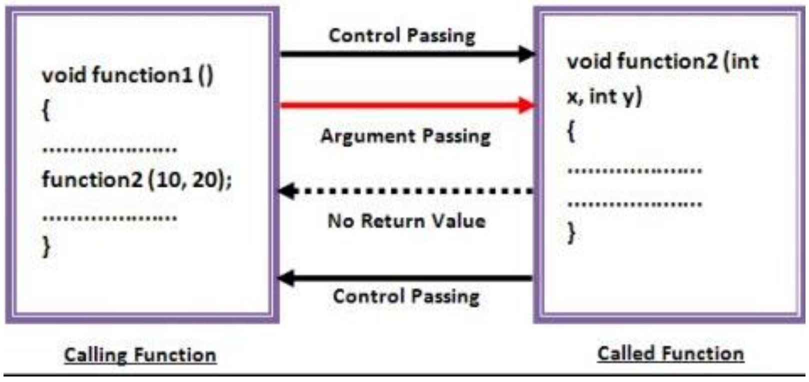
```
#include<stdio.h>
#include<conio.h>
//----------------------------------------
  void area();  // Function Declaration
//----------------------------------------
void main()
{
     clrscr();
     area();       // Function Call
     getch();
}
//----------------------------------------
void area()
{
     float ar,rad;
     printf("\nEnter the radius : ");
     scanf("%f",&rad);

     ar = 3.14 * rad * rad ;
     printf("Area of Circle = %f",ar);
}
```

- In the above example, the function **area()** doesn't take any argument nor does it return any value but still it computes the area of circle.
- The function area() receives the data (**rad**)directly from the terminal which is then computed and stored in the variable **ar** and printed on the screen using **printf()**.
- When the closing brace of **area()**is reached, the control is transferred back to the calling function **main().**
- Here no **return** statement is employed. When there is nothing to be returned, the **return** statement is optional.
- The closing brace of the function signals the end of execution of the function, thus returning the control, back to the calling function.

Functions with arguments and no return values

- We could make the calling function to read data from the terminal and pass it on to the called function.
- This approach is much better because it checks the validity of the data, if necessary, before it is handed over to the called function.

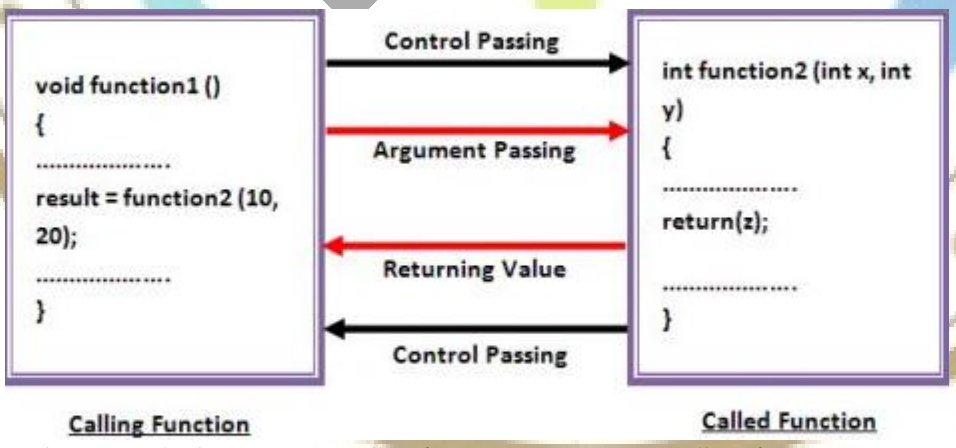


- Here the function accepts argument but it does not return a value back to the calling Program.
- It is a **Single (One-way)** Type Communication.
- Generally Output is printed in the **Called function**.
- In the example, here **area** is called function and **main** is calling function.
- Here the argument **float rad** is the formal argument. The calling function can send value to this argument using function call containing appropriate argument: **area(5)**.
- E.g.: **area(5)**; would send the 5 to function **void area(float rad)** and assign 5 to rad. The value 5 is the actual argument which becomes the value of the formal argument **rad** inside the called function.
- The actual and formal arguments must match in number, type and order. The values of actual arguments are assigned to the formal arguments on one-to-one basis, starting with the first argument.

```
#include<stdio.h>
#include<conio.h>
//------------------------------------------
  void area(float rad);  // Prototype Declaration
//------------------------------------------
void main()
{
float rad;
   printf("nEnter the radius : ");
   scanf("%f",&rad);
   area(rad);
getch();
}
//------------------------------------------
void area(float rad)
{
float ar;
ar = 3.14 * rad * rad ;
printf("Area of Circle = %f",ar);
}
```

Functions with arguments and one return values

- In some cases, the function value receives data from the calling function through arguments, but does not send back any value but it displays the results of calculations at the terminal.
- However, we may not always wish to have the result of a function displayed.
- We may use it in the calling function for further processing.
- Different programs may require different output formats for display of results.
- These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.
- Here, the function accepts a argument and it return a value back to the calling Program.
- It is Double (Two-way) Type Communication.
- Here the output is usually printed in the **main()** function.

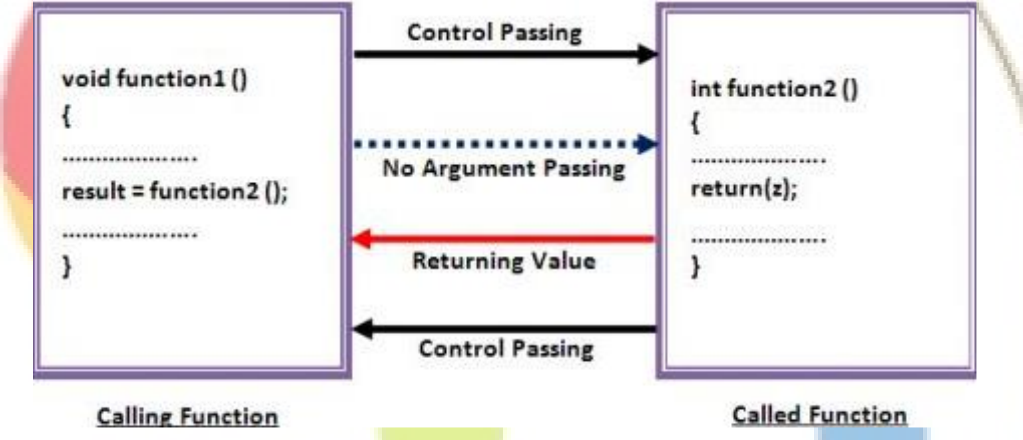


- In the example, here **area** is called function and **main** is calling function.
- Here the argument **float rad** is the formal argument. The calling function can send value to this argument using function call containing appropriate argument: **area(5)**.
- E.g.: **ar = area(5)**; would send the 5 to function **float area(float rad)** and assign 5 to rad. The value 5 is the actual argument which becomes the value of the formal argument **rad** inside the called function.
- The function is executed line by line in a normal fashion until **return(ar);** statement is reached.
- At this point, the float value of **ar** is passed back to the function-call in the **main** where the calling statement is executed and the returned value is thus assigned to **ar, a float variable** in main.

```
#include<stdio.h>
#include<conio.h>
//-----------------------------------------
  float area(float rad);  // Prototype Declaration
//-----------------------------------------
void main()
{
        float rad, ar;
        printf("nEnter the radius : ");
        scanf("%f",&rad);
        ar = area(rad);
        printf("Area of Circle = %f",ar);
getch();
}
//-----------------------------------------
float area(float rad)
{
        float ar;
        ar = 3.14 * rad * rad ;
        return ar;
}
```

Functions with no arguments but a return value

- In many situations, we may require designing functions that do not take any arguments but returns a value to the calling function.
- E.g: **getchar();** in the header file **<stdio.h>.** It accepts no parameters but returns an integer type data that represents a character.



```
#include<stdio.h>
#include<conio.h>
//-----------------------------------------
  int send();  // Prototype Declaration
//-----------------------------------------
void main()
{
        int z;
        clrscr();
        z = send();
        printf("You have entered: %d", z);
getch();
}
//-----------------------------------------
int send()
{
        int no1;
        printf("Enter a no:");
        scanf("%d",&no1);
        return (no1);
}
```

- In the above example, the function **send()** is the called function and **main()** is the calling function.
- Here the function **send()** simply returns an **int** type variable **no1** to the calling function by the execution of statement **z = send();** without accepting any parameter.
- After the execution of this statement the value returned no1 by send() is assigned to the variable z in the main().

Recursion

- When a called function in turn calls another function a process of 'chaining' occurs.
- Recursion is a special case where a function calls itself.

```
#include<stdio.h>
void main()
{
        int a, fact;
        printf("\nEnter any number: ");
        scanf ("%d", &a);
        fact=rec (a);
        printf("\nFactorial Value = %d", fact);


getch();
}
```

```
int rec (int x)
{
        int f;
        if (x==1)
        {
                return (1);
        }
        else
        {       f=x*rec(x-1);
        }
return (f);
}
```

- Here when **a** is passed as an argument to **rec(),** it will first check whether it is not 1. If it is, it returns with a value 1 which will be returned to main() and stored in **fact.** If it not 1 it will move to statement -> **f = x * rec(x – 1);** which will be evaluated for the value of **x** * **rec(x-1)** which will once again call the same function rec() taking x-1 as the parameter.
- The process will continue till the condition that x = 1 is reached.

- It will evaluated in the following sequence (for **a** = 3):

      Fact = 3 * rec(2)
           = 3 * 2 *rec(1)
           = 3 * 2 * 1 = 6
- Recursive functions are usually used for repetitive calculations.
- They can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.

Call-by-Value and Call-by-Reference

- Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called **argument**.
- There are two different methods of passing arguments to functions in C.
    1. Call-by-Value
    2. Call-by-Reference

Call-by-Value

- Call-by-value makes a copy of the variable before passing it onto a function. This means that if we try to modify the value inside a function, it will only have the modified value inside that function.
- One the function returns, the variable we passed it will have the same value it had before we passed it into the function.

```
#include<stdio.h>
void swap(int number1,int number2);
void main() {

    int num1=50,num2=70;
    swap(num1,num2);

    printf("\nNumber 1 : %d",num1);
    printf("\nNumber 2 : %d",num2);

    getch();
}
```

```
void swap(int number1,int number2)
{
    int temp;
    temp = number1;
    number1 = number2;
    number2 = temp;
}

Output:
Number 1 : 50
Number 2 : 70
```

- While Passing Parameters using call by value , **xerox copy of original parameters (num1 and num2) is created** and passed to the called function.
- Any update made inside method will not affect the **original value of variable in calling function**.
- In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1, number2 variable of sum function respectively.
- As their scope is limited to only function so they **cannot alter the values inside main function**.

Call-by-Reference

- When a function is called by the reference then the values those are passed in the calling functions are affected when they are passed by reference means they change their value when they passed by the References.

- In the Call by Reference we pass the Address of the variables whose Arguments are also Send. So that when we use the Reference then, we pass the Address the Variables.

```
#include<stdio.h>
void swap(int *number1,int *number2);

void main() {
    int num1=50,num2=70;
    swap(&num1,&num2);

    printf("\nNumber 1 : %d",num1);
    printf("\nNumber 2 : %d",num2);

    return(0);
}
```

```
void swap(int *num1,int *num2)
{
    int temp;
    temp  = *num1;
    *num1 = *num2;
    *num2 = *num1;
}
```

```
Output:
Number 1 : 70
Number 2 : 50
```

- While passing parameter using call by address scheme, we are passing the actual address of the variable to the called function (**here &num1 and &num2**).
- Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.

Call by Value	Call by Reference
This is the usual method to call a function in which only the **value of the variable** is passed as an argument	In this method, the **address of the variable** is passed as an argument
Any alternation in the value of the argument passed is local to the function and **is not accepted** in the calling program	Any alternation in the value of the argument passed **is accepted** in the calling program(since alternation is made indirectly in the memory location using the pointer)
Memory location occupied by formal and actual arguments is **different**	Memory location occupied by formal and actual arguments is **same** and there is a saving of memory location
Since a new location is created, this method is **slow**	Since the existing memory location is used through its address, this method is **fast**
There is **no possibility of wrong data manipulation** since the arguments are directly used in an application	There is a **possibility of wrong data manipulation** since the addresses are used in an expression.